Lab Manual
for
Linear Algebra
by
Jim Hefferon

Preface

This collection supplements the text *Linear Algebra*[1] with a number of explorations that help students solidify and extend their understanding of the subject, using the mathematical software Sage.[2]

A major goal of any undergraduate Mathematics program is to, over time, move students toward a higher-level, more abstract, grasp of the subject. For instance, most Calculus classes work on elaborate computations while later courses spend more effort on concepts and proofs, leaving less for the details of calculations. The text *Linear Algebra* fits into this development process.

Naturally the text presents its material using examples and practice problems that are small-sized and have manageable numbers: an assignment to multiply a pair of three by three matrices of small integers will build intuition, whereas asking students to do that same by-hand question with twenty by twenty matrices of ten decimal place numbers would be badgering. (And, even more worrisome, having students focus their intellectual energy on calculations instead of directing their attention to the ideas and proofs misleads them as to what the subject is about.)

However, mathematical software can mitigate this by extending the reach of what is reasonable to bigger systems, harder numbers, and computations that — while too much to do by hand — yield interesting information when they are done by a machine. For instance, an advantage of learning how to handle these tougher computations is that they are more like the ones that appear when students apply Linear Algebra to other subjects. Another advantage is that students see new ideas such as runtime growth measures.

Well then, why not teach straight from the computer system? Remember that we want to develop a higher-level understanding of the material, keeping the focus on vector spaces and linear maps. In our development the computations are a way to develop that understanding, not the main point. Some instructors may find that for their students this work is best left aside altogether, keeping a tight focus on the core material, while other instructors have students who will benefit.

Why Sage?

Sage is a very powerful mathematical software systems but so are many others. This manual uses it because it is Free[3] and Open Source[4] software.

In *Open Source Mathematical Software* [Joyner and Stein, 2007][5] the authors argue that for Mathematics the best way forward is to use software that is Open Source.

---

[1]The text's home page http://joshua.smcvt.edu/linearalgebra has the PDF, the ancillary materials, and the LaTeX source. [2]See http://www.sagemath.org for the software and documentation. [3]The Free Software Foundation page http://www.gnu.org/philosophy/free-sw.html gives background and a definition. [4]See http://opensource.org/osd.html for a definition. [5]See http://www.ams.org/notices/200710/tx071001279p.pdf for the full text.

Suppose Jane is a well-known mathematician who announces she has proved a theorem. We probably will believe her, but she knows that she will be required to produce a proof if requested. However, suppose now Jane says a theorem is true based partly on the results of software. The closest we can reasonably hope to get to a rigorous proof (without new ideas) is the open inspection and ability to use all the computer code on which the result depends. If the program is proprietary, this is not possible. We have every right to be distrustful, not only due to a vague distrust of computers but because even the best programmers regularly make mistakes.

If one reads the proof of Jane's theorem in hopes of extending her ideas or applying them in a new context, it is limiting to not have access to the inner workings of the software on which Jane's result builds.

Professionals choose their tools by balancing many factors but their argument is persuasive. This manual uses Sage because it is very capable, including at Linear Algebra, because students can learn a great deal from it, and because it is Free.

This manual

This is Free. Get the latest version from http://joshua.smcvt.edu/linearalgebra. Also see that page for the license details and for the LaTeX source, including this manual.

I am glad to hear suggestions or corrections, especially from instructors who have class-tested the material. My contact information is on the same page.

The Sage output in this manual was generated automatically so it is sure to be accurate, except that I have (automatically) edited a few lines for length. My Sage identifies itself in this way.

```
1  'Sage Version 5.2, Release Date: 2012-07-25'
```

Acknowledgements

I am glad for this chance to thank the Sage Development Team for their work. In particular, without [Sage Development Team, 2012b] this manual would not have happened. I am glad also for the chance to mention [Beezer, 2011] as an inspiration.

Jim Hefferon
Mathematics, Saint Michael's College
Colchester, Vermont USA
2012-Sep-10

Contents

Python and Sage

To work through the Linear Algebra in this manual you need to be acquainted with how to run Sage. Sage uses the computer language Python so we'll start with that.

Python

Python is a popular computer language, often used for scripting, that is appealing for its simple style and powerful libraries. The significance for us of 'scripting' is that Sage uses it in this way, as a glue to bring together separate parts.

Python is Free. If your operating system doesn't provide it then go to the home page www.python.org and follow the download and installation instructions. Also at that site is Python's excellent tutorial. That tutorial is thorough; here you will see only enough Python to get started.

Comment. There is a new version, Python 3, with some differences. Here we stick to the older version because that is what Sage uses.

Basics Start Python, for instance by entering `python` at a command line. You'll get a couple of lines of information followed by three greater-than characters.

```
>>>
```

This is a prompt. It lets you experiment: if you type Python code and ⟨*Enter*⟩ then the system will read your code, evaluate it, and print the result. We will see below how to write and run whole programs but for now we will experiment. You can always leave the prompt with ⟨*Ctrl*⟩-*D*.

Try entering these expressions (double star is exponentiation).

```
>>> 2 - (-1)
3
>>> 1 + 2*3
7
>>> 2**3
8
```

Part of Python's appeal is that simple things tend to be simple to do. Here is how you print something to the screen.

```
>>> print 1, "plus", 2, "equals", 3
1 plus 2 equals 3
```

Often you can debug just by putting in commands to print things, and having a straightforward print operator helps with that.

As in any other computer language, variables give you a place to keep values.

```
1  >>> i = 1
2  >>> i + 1
3  2
```

In some programming languages you must declare the 'type' of a variable before you use it; for instance you would have to declare that i is an integer before you could set $i = 1$. In contrast, Python deduces the type of a variable based on what you do to it — above we assigned 1 to i so Python figured that it must be an integer. Further, we can change how we use the variable and Python will go along; here we change what is in x from an integer to a string.

```
1  >>> x = 1
2  >>> x
3  1
4  >>> x = 'a'
5  >>> x
6  'a'
```

Python complains by raising an *error*. Here we are trying to combine a string and an integer.

```
1  >>> 'a'+1
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in <module>
4  TypeError: cannot concatenate 'str' and 'int' objects
```

The error message's bottom line is the useful one.

The hash mark # makes the rest of a line a comment.

```
1  >>> t = 2.2
2  >>> d = (0.5) * 9.8 * (t**2)   # d in meters
3  >>> d
4  23.716000000000005
```

(Comments are more useful in a program than at the prompt.) Programmers often comment an entire line by starting that line with a hash.

As in the listing above, we can represent real numbers and even complex numbers.

```
1  >>> 5.774 * 3
2  17.322
3  >>> (3+2j) - (1-4j)
4  (2+6j)
```

Notice that Python uses 'j' for the square root of -1, not the 'i' traditional in mathematics.

The examples above show addition, subtraction, multiplication, and exponentiation. Division is a bit awkward. Python was originally designed to have the division bar / mean real number division when at least one of the numbers is real. However between two integers the division bar was taken to mean a quotient, as in "2 goes into 5 with quotient 2 and remainder 1."

```
1  >>> 5.2 / 2.0
2  2.6
3  >>> 5.2 / 2
4  2.6
5  >>> 5 / 2
6  2
```

This was a mistake and one of the changes in Python 3 is that the quotient operation will be $//$ while the single-slash operator will be real division in all cases. In Python 2 the simplest thing is to make sure that at least one number in a division is real.

```
1  >>> x = 5
2  >>> y = 2
3  >>> (1.0*x) / y
4  2.5
```

Incidentally, do the remainder operation (sometimes called 'modulus') with a percent character: 5 % 2 returns 1.

Variables can also represent truth values; these are *Booleans*.

```
1  >>> yankees_stink = True
2  >>> yankees_stink
3  True
```

You need the initial capital: `True` or `False`, not `true` or `false`.

Above we saw a string consisting of text between single quotes. You can use either single quotes or double quotes, as long as you use the same at both ends of the string. Here `x` and `y` are double-quoted, which makes sense because they contain apostrophes.

```
1  >>> x = "I'm Popeye the sailor man"
2  >>> y = "I yam what I yam and that's all what I yam"
3  >>> x + ', ' + y
4  "I'm Popeye the sailor man, I yam what I yam and that's all what I yam"
```

The + operation concatenates strings. Inside a double-quoted string you can use slash-n `\n` to get a newline.

A string marked by three sets of quotes can contain line breaks.

```
1   >>> """THE ROAD TO WISDOM
2   ...
3   ... The road to wisdom?
4   ... -- Well, it's plain
5   ... and simple to express:
6   ... Err
7   ... and err
8   ... and err again
9   ... but less
10  ... and less
11  ... and less. --Piet Hein"""
```

The triple dots at the start of each line is a prompt that Python's read-eval-print loop gives when what you have typed is not complete. One use for triple-quoted strings is as documentation in a program; we'll see that below.

A Python *dictionary* is a finite function, that is, it is a finite set of ordered pairs ⟨key, value⟩ subject to the restriction that no key can be repeated.

```
1  >>> english_words = {'one': 1, 'two': 2, 'three': 3}
2  >>> english_words['one']
3  1
4  >>> english_words['four'] = 4
```

Dictionaries are a simple database. But the fact that a dictionary is a set means that the elements come in no apparently-sensible order.

```
1 >>> english_words
2 {'four': 4, 'three': 3, 'two': 2, 'one': 1}
```

(Don't be mislead by this example, they do not always just come in the reverse of the order in which you entered them.) If you assign to an existing key then that will replace the previous value.

```
1 >>> english_words['one'] = 5
2 >>> english_words
3 {'four': 4, 'three': 3, 'two': 2, 'one': 5}
```

Dictionaries are central to Python, in part because looking up values in a dictionary is very fast.

While dictionaries are unordered, a Python *list* is ordered.

```
1 >>> a=['alpha', 'beta', 'gamma']
2 >>> b=[]
3 >>> c=['delta']
4 >>> a
5 ['alpha', 'beta', 'gamma']
6 >>> a+b+c
7 ['alpha', 'beta', 'gamma', 'delta']
```

Get an element from a list by specifying its index, its place in the list, inside square brackets. Lists are zero-offset indexed, that is, the initial element of the list is numbered 0. You can count from the back by using negative indices.

```
1 >>> a[0]
2 'alpha'
3 >>> a[1]
4 'beta'
5 >>> a[-1]
6 'gamma'
```

Also, specifying two indices separated by a colon will get a *slice* of the list.

```
1 >>> a[1:3]
2 ['beta', 'gamma']
3 >>> a[1:2]
4 ['beta']
```

You can add to a list.

```
1 >>> c.append('epsilon')
2 >>> c
3 ['delta', 'epsilon']
```

Lists can contain anything, including other lists.

```
1 >>> x = 4
2 >>> a = ['alpha', ['beta', x]]
3 >>> a
4 ['alpha', ['beta', 4]]
```

The function `range` returns a list of numbers.

```
1    >>> range (10)
2  [0,  1,  2,  3,  4,  5,  6,  7,  8,  9]
3  >>> range (1 ,10)
4  [1,  2,  3,  4,  5,  6,  7,  8,  9]
```

Observe that by default `range` starts at 0, which is convenient because lists are zero-indexed. Observe also that 9 is the highest number in the sequence given by `range(10)`. This makes `range(10)+range(10,20)` give the same list as `range(20)`.

One of the most common things you'll do with a list is to run through it performing some action on each entry. Python has a shortcut for this called *list comprehension*.

```
1  >>> a  =  [2** i  for  i  in  range (4)]
2  >>> a
3  [1,  2,  4,  8]
4  >>> [i -1  for  i  in  a]
5  [0,  1,  3,  7]
```

This is a special form of a loop; we'll see more on loops below.

A *tuple* is like a list in that it is ordered.

```
1  >>> a  =  (0,  1,  2)
2  >>> a
3  (0,  1,  2)
4  >>> a [0]
5  0
```

However, unlike a list a tuple cannot change.

```
1  >>> a [0]  =  3
2  Traceback (most  recent  call  last ):
3    File  "<stdin >",  line  1,  in  <module >
4  TypeError:  'tuple '  object  does  not  support  item  assignment
```

That is an advantage sometimes: because tuples cannot change they can be keys in dictionaries, while list cannot.

```
1  >>> a  =  ['Jim ',  2138]
2  >>> b  =  ('Jim ',  2138)
3  >>> d  =  {a:  1}
4  Traceback (most  recent  call  last ):
5    File  "<stdin >",  line  1,  in  <module >
6  TypeError:  unhashable  type:  'list '
7  >>> d  =  {b:  1}
8  >>> d
9  {('Jim ',  2138):  1}
```

Python has a special value `None` for you to use where there is no sensible value. For instance, if your program keeps track of a person's address and includes a variable `appt_no` then `None` is the right value for that variable when the person does not live in an apartment.

Flow of control   Python supports the traditional ways of affecting the order of statement execution, with a twist.

```
1  >>> x  =  4
2  >>> if  (x  ==  0):
```

```
3  ...          y  =  1
4  ...     else:
5  ...          y  =  0
6  ...
7  >>>  y
8  0
```

The twist is that while many languages use braces or some other syntax to mark a block of code, Python uses indentation. (We shall always indent with four spaces.) Here, Python executes the single-line block y = 1 if x equals 0, otherwise Python sets y to 0.

Notice also that double equals == means "is equal to." We have already seen that single equals is the assignment operation so that x = 4 means "x is assigned the value 4."

Python has two variants on the if statement. The first has only one branch

```
1  >>>  x  =  4
2  >>>  y  =  0
3  >>>  if  (x  ==  0):
4  ...          y  =  1
5  ...
6  >>>  y
7  0
```

while the second has more than two branches.

```
1  >>>  x  =  2
2  >>>  if  (x  ==  0):
3  ...          y  =  1
4  ...     elif  (x  ==  1):
5  ...          y  =  0
6  ...     else:
7  ...          y  =  -1
8  ...
9  >>>  y
10  -1
```

A specialty of computers is iteration, looping through the same steps.

```
1  >>>  for  i  in  range(5):
2  ...          print  i,  "squared  is",  i**2
3  ...
4  0  squared  is  0
5  1  squared  is  1
6  2  squared  is  4
7  3  squared  is  9
8  4  squared  is  16
```

A for loop often involves a range.

```
1  >>>  x=[4,0,3,0]
2  >>>  for  i  in  range(len(x)):
3  ...          if  (x[i]  ==  0):
4  ...                print  "item",i,"is  zero"
5  ...          else:
6  ...                print  "item",i,"is  nonzero"
7  ...
```

```
8   item 0 is nonzero
9   item 1 is zero
10  item 2 is nonzero
11  item 3 is zero
```

We could instead have written `for c in x:` since the `for` loop can iterate over any sequence, not just a sequence of integers.

A `for` loop is designed to execute a certain number of times. The natural way to write a loop that will run an uncertain number of times is `while`.

```
1   >>> n = 27
2   >>> i = 0
3   >>> while (n != 1):
4   ...        if (n%2 == 0):
5   ...            n = n / 2
6   ...        else:
7   ...            n = 3*n + 1
8   ...        i = i + 1
9   ...        print "i=", i
10  ...
11  i= 1
12  i= 2
13  i= 3
```

(This listing is incomplete; it takes 111 steps to finish.)[1] Note that "not equal" is `!=`.

The `break` command gets you out of a loop right away.

```
1   >>> for i in range(10):
2   ...        if (i == 3):
3   ...            break
4   ...        print "i is", i
5   ...
6   i is 0
7   i is 1
8   i is 2
```

Functions A *function* is a group of statements that executes when it is called, and can return values to the caller. Here is a naive version of the quadratic formula.

```
1   >>> def quad_formula(a, b, c):
2   ...        discriminant = (b**2 - 4*a*c)**(0.5)
3   ...        r1=(-1*b+discriminant) / (2.0*a)
4   ...        r2=(-1*b-discriminant) / (2.0*a)
5   ...        return (r1, r2)
6   ...
7   >>> quad_formula(1,0,-9)
8   (3.0, -3.0)
9   >>> quad_formula(1,2,1)
10  (-1.0, -1.0)
```

---

[1] The *Collatz conjecture* is that for any starting n this loop will terminate, but this is not known.

(One way that it is naive is that it doesn't handle complex roots gracefully.) Functions organize code into blocks that may be run a number of different times or which may belong together conceptually. In a Python program the great majority of code is in functions.

At the end of the `def` line, in parentheses, are the function's *parameters*. These can take values passed in by the caller. Functions can have *optional parameters* that have a default value.

```
1  >>> def hello(name="Jim"):
2  ...       print "Hello,", name
3  ...
4  >>> hello("Fred")
5  Hello, Fred
6  >>> hello()
7  Hello, Jim
```

Sage uses this aspect of Python a great deal.

Functions can contain multiple `return` statements. They always return something; if a function never executes a `return` then it will return the value `None`.

Objects and modules In Mathematics, the real numbers is a set associated with some operations such as addition and multiplication. Python is *object-oriented*, which means that we can similarly bundle together data and actions.

```
1  >>> class person(object):
2  ...       def __init__(self, name, age):
3  ...           self.name = name
4  ...           self.age = age
5  ...       def hello(self):
6  ...           print "Hello", self.name
7  ...
8  >>> a=person("Jim", 53)
9  >>> a.hello()
10 Hello Jim
11 >>> a.age
12 53
```

You work with objects by using the *dot notation*: to get the age data bundled with `a` you write `a.age`, and to call the `hello` function bundled with `a` you write `a.hello()` (a function bundled in this way is called a *method*).

You won't be writing your own classes in this lab manual but you will be using ones from the extensive libraries of code that others have written, including the code for Sage. For instance, Python has a library, or *module*, for math.

```
1  >>> import math
2  >>> math.pi
3  3.141592653589793
4  >>> math.factorial(4)
5  24
6  >>> math.cos(math.pi)
7  -1.0
```

The `import` statement gets the module and makes its contents available.

Another library is for random numbers.

```
1  >>> import random
2  >>> while (random.randint(1,10) != 1):
3  ...       print "wrong"
4  ...
5  wrong
6  wrong
```

Programs   The read-eval-print loop is great for small experiments but for more than four or five lines you want to put your work in a separate file and run it as a standalone program.

To write the code, use a text editor; one example is Emacs[1]. You should try to use an editor with support for Python such as automatic indentation, and syntax highlighting, where the editor colors your code to make it easier to read.

Here is one example. Start your editor, open a file called test.py, and enter these lines. Note the triple-quoted documentation string at the top of the file; good practice is to include this documentation in everything you write.

```
1  # test.py
2  """test
3
4  A test program for Python.
5  """
6
7  import datetime
8
9  current = datetime.datetime.now()  # get a datetime object
10 print "the month number is", current.month
```

Run it under Python (for instance, from the command line run python test.py) and you should see output like the month number is 9.

Here is a small game (it has some Python constructs that you haven't seen but that are straight-forward).

```
1  # guessing_game.py
2  """guessing_game
3
4  A toy game for demonstration.
5  """
6  import random
7  CHOICE = random.randint(1,10)
8
9  def test_guess(guess):
10     """Decide if the guess is correct and print a message.
11     """
12     if (guess < CHOICE):
13         print "  Sorry, your guess is too low"
14         return False
15     elif (guess > CHOICE):
16         print "  Sorry, your guess is too high"
17         return False
18     print "  You are right!"
```

---

[1]It may come with your operating system or see http://www.gnu.org/software/emacs.

```
19      return True
20
21 flag = False
22 while (not flag):
23      guess = int(raw_input("Guess an integer between 1 and 10: "))
24      flag = test_guess(guess)
```

Here is the output.

```
1 $ python guessing_game.py
2 Guess an integer between 1 and 10: 5
3   Sorry, your guess is too low
4 Guess an integer between 1 and 10: 8
5   Sorry, your guess is too high
6 Guess an integer between 1 and 10: 6
7   Sorry, your guess is too low
8 Guess an integer between 1 and 10: 7
9   You are right!
```

As above, note the triple-quoted documentation strings both for the file as a whole and for the function. Go to the directory containing guessing_game.py and start Python. At the prompt type in import guessing_game. You will play through a round of the game (there is a way to avoid this but it doesn't matter here) and then type help("guessing_game"). You will see documentation that includes these lines.

```
1 DESCRIPTION
2     A toy game for demonstration.
3
4 FUNCTIONS
5     test_guess(guess)
6         Decide if the guess is correct and print a message.
```

Obviously, Python got this from the file's documentation strings. In Python, and also in Sage, good practice is to always include documentation that is accessible with the help command.

Sage

Learning what Sage can do is the object of much of this book so this is a very brief walk-through of preliminaries. See [Sage Development Team, 2012a] for a more broad-based introduction.

First, if your system does not already supply it then install Sage by following the directions at www.sagemath.org.

Command line   Sage's command line is like Python's but adapted to mathematical work. First start Sage, for instance, enter sage into a command line window. You get some initial text and then a prompt (leave the prompt by typing exit and ⟨*Enter*⟩.)

```
1 sage:
```

Experiment with some expressions.

```
1 sage: 2**3
2 8
```

```
3  sage: 2^3
4  8
5  sage: 3*1 + 4*2
6  11
7  sage: 5 == 3+3
8  False
9  sage: sin(pi/3)
10 1/2*sqrt(3)
```

The second expression shows that Sage provides a convenient shortcut for exponentiation. The fourth shows that Sage sometimes returns exact results rather than an approximation. You can still get the approximation.

```
1  sage: sin(pi/3).numerical_approx()
2  0.866025403784439
3  sage: sin(pi/3).n()
4  0.866025403784439
```

The function n() is an abbreviation for numerical_approx().

Script You can group Sage commands together in a file. This way you can test the commands, and also reuse them without having to retype.

Create a file with the extension .sage, such as sage_try.sage. Enter this function and save the file.

```
1  def normal_curve(upper_limit):
2      """Approximate area under the Normal curve from 0 to upper_limit.
3      """
4      stdev = 1.0
5      mu = 0.0
6      area=numerical_integral((1/sqrt(2*pi) * e^(-0.5*(x)^2)),
7                              0, upper_limit)
8      print "area is", area[0]
```

Bring in the commands in with a load command.

```
1  sage: load "sage_try.sage"
2  sage: normal_curve(1.0)
3  area is 0.341344746069
```

Notebook Sage also offers a browser-based interface, where you can set up worksheets to run alone or with other people, where you can easily view plots integrated with the text, and many other nice features.

From the Sage prompt run notebook() and work through the tutorial.

Gauss's Method

Sage can solve linear systems in a number of ways. Here you will enter systems of linear equations, both as equations and in the matrix form. You will use those to solve the systems.

Systems

To enter a system of equations you must first enter single equations, so you must enter variables. We have seen one kind of variable in giving commands like these.

```
sage: x = 3
sage: 7*x
21
```

Here x is the name of a location to hold values. For variables in equations we want something different; in the equation $C = 2\pi \cdot r$ the two variables do not necessarily have fixed values, nor are they associated with a location in the computer's memory. To illustrate the difference enter this.

```
sage: y
```

The response is `NameError: name 'y' is not defined` because Sage defaults to expecting that y is the name of a location that holds a value. Instead, you need to declare to the system that something is a symbolic variable.

```
sage: var('x,y')
sage: x
x
```

With that, a system of equations is a list.

```
sage: var('x,y,z')
(x, y, z)
sage: eqns = [x-y+2*z == 4, 2*x+2*y == 12, x-4*z==5]
sage: solve(eqns, x,y,z)
[[x == 5, y == 1, z == 0]]
```

Note that you must write double equals `==` instead of the assignment operator `=` and also that you must write `2*x` instead of `2x`. Either mistake will give you `SyntaxError: invalid syntax`.

You can put a parameter in the right side and solve for the variables in terms of the parameter.

```
sage: var('x,y,z,a')
(x, y, z, a)
sage: eqns = [x-y+2*z == a, 2*x+2*y == 12, x-4*z==5]
sage: solve(eqns, x,y,z)
[[x == 2/5*a + 17/5, y == -2/5*a + 13/5, z == 1/10*a - 2/5]]
```

Matrices The *solve* routine is general-purpose but for the special case of linear systems matrix notation is the best tool.

Most of your matrices have entries that are fractions. Sage uses 'QQ' for the rational numbers, 'RR' for the reals, 'CC' for the complex numbers, and 'ZZ' for the integers. Also, you enter a row of the matrix as a list, and you enter the entire matrix as a list of rows.

```
1  sage: M = matrix(QQ, [[1, 2, 3], [4, 5, 6], [7 ,8, 9]])
2  sage: M
3  [1 2 3]
4  [4 5 6]
5  [7 8 9]
6  sage: M[1,2]
7  6
8  sage: M.nrows()
9  3
10 sage: M.ncols()
11 3
```

Sage lists are zero-indexed, so M[1,2] asks for the entry in the second row and third column.

Enter a vector in much the same way.

```
1  sage: v = vector(QQ, [2/3, -1/3, 1/2])
2  sage: v
3  (2/3, -1/3, 1/2)
4  sage: v[1]
5  -1/3
```

This is a column vector; it appears with rounded brackets so that it looks different than a one-row matrix. Augment the matrix with this vector.

```
1  sage: M_prime = M.augment(v)
2  sage: M_prime
3  [ 1  2  3 10]
4  [ 4  5  6 11]
5  [ 7  8  9 12]
```

Have Sage remember the distinction between the two parts of M'.

```
1  sage: M_prime = M.augment(v, subdivide=True)
2  sage: M_prime
3  [ 1  2  3|10]
4  [ 4  5  6|11]
5  [ 7  8  9|12]
```

Row operations Computers are good for jobs that are tedious and error-prone. Row operations are both.

First enter an example matrix.

```
1  sage: M = matrix(QQ, [[0, 2, 1], [2, 0, 4], [2 ,-1/2, 3]])
2  sage: v = vector(QQ, [2, 1, -1/2])
3  sage: M_prime = M.augment(v, subdivide=True)
4  sage: M_prime
5  [   0    2    1|   2]
6  [   2    0    4|   1]
7  [   2 -1/2    3|-1/2]
```

Swap the top rows (remember that row indices start at zero).

```
1  sage:  M_prime.swap_rows(0,1)
2  sage:  M_prime
3  [    2      0      4|    1]
4  [    0      2      1|    2]
5  [    2   -1/2      3|-1/2]
```

Rescale the top row.

```
1  sage:  M_prime.rescale_row(0,  1/2)
2  sage:  M_prime
3  [    1      0      2|  1/2]
4  [    0      2      1|    2]
5  [    2   -1/2      3|-1/2]
```

Get a new bottom row by adding -2 times the top to the current bottom row.

```
1  sage:  M_prime.add_multiple_of_row(2,0,-2)
2  sage:  M_prime
3  [    1      0      2|  1/2]
4  [    0      2      1|    2]
5  [    0   -1/2     -1|-3/2]
```

Finish by finding echelon form.

```
1  sage:  M_prime.add_multiple_of_row(2,1,1/4)
2  sage:  M_prime
3  [    1      0      2|  1/2]
4  [    0      2      1|    2]
5  [    0      0   -3/4|   -1]
```

(By the way, Sage would have given us echelon form in a single operation had we run the command M_prime.echelon_form().)

Now by-hand back substitution gives the solution, or we can use solve.

```
1  sage:  var('x,y,z')
2  sage:  eqns=[-3/4*z  ==  -1,  2*y+z  ==  2,  x+2*z  ==  1/2]
3  sage:  solve(eqns,  x,  y,  z)
4  [[x  ==  (-13/6),  y  ==  (1/3),  z  ==  (4/3)]]
```

The operations swap_rows, rescale_rows, and add_multiple_of_row changed the matrix M'. Sage has related commands that return a changed matrix but leave the starting matrix unchanged.

```
1  sage:  M  =  matrix(QQ,  [[1/2,  1,  -1],  [1,  -2,  0],  [2 ,-1,  1]])
2  sage:  v  =  vector(QQ,  [0,  1,  -2])
3  sage:  M_prime  =  M.augment(v,  subdivide=True)
4  sage:  M_prime
5  [1/2    1   -1|   0]
6  [  1   -2    0|   1]
7  [  2   -1    1|  -2]
8  sage:  N  =  M_prime.with_rescaled_row(0,2)
9  sage:  M_prime
10 [1/2    1   -1|   0]
11 [  1   -2    0|   1]
```

```
12 [   2   -1    1|  -2]
13 sage: N
14 [  1   2  -2|  0]
15 [  1  -2   0|  1]
16 [  2  -1   1|-2]
```

Here, M' is unchanged by the routine, while N is the returned changed matrix. The other two routines of this kind are `with_swapped_rows` and `with_added_multiple_of_row`.

Nonsingular and singular systems   Resorting to `solve` after going through the row operations is artless. Sage will give reduced row echelon form straight from the augmented matrix.

```
1 sage: M_prime.rref()
2 [     1      0      0| -13/6]
3 [     0      1      0|   1/3]
4 [     0      0      1|   4/3]
```

In that example the matrix M on the left is nonsingular because it is square and because Gauss's Method leads to echelon forms where every one of M's columns has a leading variable. The next example starts with a different square matrix, a singular matrix, and consequently leads to echelon form systems where there are columns on the left that do not have a leading variable.

```
1 sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2 , 3, 4]])
2 sage: v = vector(QQ, [0, 1, 1])
3 sage: M_prime = M.augment(v, subdivide=True)
4 sage: M_prime
5 [1  1  1|0]
6 [1  2  3|1]
7 [2  3  4|1]
8 sage: M_prime.rref()
9 [ 1   0  -1|-1]
10 [ 0   1   2| 1]
11 [ 0   0   0| 0]
```

Recall that the singular case has two subcases. The first happens above: because in echelon form every row that is all zeros on the left has an entry on the right that is also zero, the system has infinitely many solutions. In contrast, with the same starting matrix the example below has a row that is zeros on the left but is nonzero on the right and so the system has no solution.

```
1 sage: v = vector(QQ, [0, 1, 2])
2 sage: M_prime = M.augment(v, subdivide=True)
3 sage: M_prime.rref()
4 [ 1   0  -1|  0]
5 [ 0   1   2|  0]
6 [ 0   0   0|  1]
```

The difference between the subcases has to do with the relationships among the rows of M and the relationships among the rows of the vector. In both cases, the relationship among the rows of the matrix M is that the first two rows add to the third. In the first case the vector has the same relationship while in the second it does not.

The easy way to ensure that a zero row in the matrix on the left is associated with a zero entry in the vector on the right is to make the vector have all zeros, that is, to consider the homogeneous system associated with M.

```
1  sage: v = zero_vector(QQ, 3)
2  sage: v
3  (0, 0, 0)
4  sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2, 3, 4]])
5  sage: M_prime = M.augment(v, subdivide=True)
6  sage: M_prime
7  [1 1 1|0]
8  [1 2 3|0]
9  [2 3 4|0]
10 sage: M_prime.rref()
11 [ 1  0 -1| 0]
12 [ 0  1  2| 0]
13 [ 0  0  0| 0]
```

You can get the numbers of the columns having leading entries with the `pivots` method (there is a complementary `nonpivots`).

```
1  sage: v = vector(QQ, [0, 1, 2])
2  sage: M_prime
3  [1 1 1|0]
4  [1 2 3|1]
5  [2 3 4|2]
6  sage: M_prime.pivots()
7  (0, 1, 3)
8  sage: M_prime.rref()
9  [ 1  0 -1| 0]
10 [ 0  1  2| 0]
11 [ 0  0  0| 1]
```

Because column 2 is not in the list of pivots we know that the system is singular before we see it in echelon form.

We can use this observation to write a routine that decides if a square matrix is nonsingular.

```
1  sage: def check_nonsingular(mat):
2  ....:     if not(mat.is_square()):
3  ....:         print "ERROR: mat must be square"
4  ....:         return
5  ....:     p = mat.pivots()
6  ....:     for col in range(mat.ncols()):
7  ....:         if not(col in p):
8  ....:             print "nonsingular"
9  ....:             break
10 ....:
11 sage: N = Matrix(QQ, [[1, 2, 3], [4, 5, 6], [7, 8, 9]])
12 sage: check_nonsingular(N)
13 nonsingular
14 sage: N = Matrix(QQ, [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
15 sage: check_nonsingular(N)
```

Actually, Sage matrices already have a method `is_singular` but this illustrates how you can write routines to extend Sage.

Parametrization   You can use `solve` to give the solution set of a system with infinitely many solutions. Start with the square matrix of coefficients from above, where the top two rows add to the bottom row, and adjoin a vector satisfying the same relationship to get a system with infinitely many solutions. Then convert that to a system of equations.

```
sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2 , 3, 4]])
sage: v = vector(QQ, [1, 0, 1])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1  1  1|1]
[1  2  3|0]
[2  3  4|1]
sage: eqns = [x+y+z == 1, x+2*y+3*z == 0, 2*x+3*y+4*z == 1]
sage: solve(eqns, x, y)
[[x == z + 2, y == -2*z - 1]]
sage: solve(eqns, x, y, z)
[[x == r1 + 2, y == -2*r1 - 1, z == r1]]
```

The first of the two `solve` calls asks Sage to solve only for x and y and so the solution is in terms of z. In the second call Sage produces a parameter of its own.

Automation

We finish by showing two routines to automate the by-hand row reductions of the kind that the text has in the homework. These use the matrix capabilities of Sage to both describe and perform the row operations that bring a matrix to echelon form or to reduced echelon form.

Loading and running   The source file of the script is below, at the end. First here are a few sample calls. Start Sage in the directory containing the file `gauss_method.sage`.

```
sage: load "gauss_method.sage"
sage: M = matrix(QQ, [[1/2, 1, 4], [2, 4, -1], [1, 2, 0]])
sage: v = vector(QQ, [-2, 5, 4])
sage: M_prime = M.augment(v, subdivide=True)
sage: gauss_method(M_prime)
[1/2   1   4| -2]
[  2   4  -1|  5]
[  1   2   0|  4]
 take -4 times row 1 plus row 2
 take -2 times row 1 plus row 3
[1/2   1   4| -2]
[  0   0 -17| 13]
[  0   0  -8|  8]
 take -8/17 times row 2 plus row 3
[  1/2     1      4|    -2]
[    0     0    -17|    13]
[    0     0      0|32/17]
```

The remaining examples skip the extra steps to make an augmented matrix.

```
sage: M1 = matrix(QQ, [[2, 0, 1, 3], [-1, 1/2, 3, 1], [0, 1, 7, 5]])
```

```
2  sage:  gauss_method(M1)
3  [   2    0    1    3]
4  [  -1  1/2    3    1]
5  [   0    1    7    5]
6   take  1/2  times  row  1  plus  row  2
7  [   2    0    1    3]
8  [   0  1/2  7/2  5/2]
9  [   0    1    7    5]
10  take  -2  times  row  2  plus  row  3
11 [   2    0    1    3]
12 [   0  1/2  7/2  5/2]
13 [   0    0    0    0]
```

The script has a routine to go all the way to reduced echelon form.

```
1  sage:  M2  =  matrix(QQ,
2  ....:                   [[1,  2,  3,  4],  [1,  2,  3,  4],
3  ....:                    [2,  4,  -1,  5],  [1,  2,  0,  4]])
4  sage:  gauss_jordan(M2)
5  [  1    2    3    4]
6  [  1    2    3    4]
7  [  2    4   -1    5]
8  [  1    2    0    4]
9   take  -1  times  row  1  plus  row  2
10  take  -2  times  row  1  plus  row  3
11  take  -1  times  row  1  plus  row  4
12 [  1    2    3    4]
13 [  0    0    0    0]
14 [  0    0   -7   -3]
15 [  0    0   -3    0]
16  swap  row  2  with  row  3
17 [  1    2    3    4]
18 [  0    0   -7   -3]
19 [  0    0    0    0]
20 [  0    0   -3    0]
21  take  -3/7  times  row  2  plus  row  4
22 [   1    2    3    4]
23 [   0    0   -7   -3]
24 [   0    0    0    0]
25 [   0    0    0  9/7]
26  swap  row  3  with  row  4
27 [   1    2    3    4]
28 [   0    0   -7   -3]
29 [   0    0    0  9/7]
30 [   0    0    0    0]
31  take  -1/7  times  row  2
32  take  7/9  times  row  3
33 [   1    2    3    4]
34 [   0    0    1  3/7]
35 [   0    0    0    1]
36 [   0    0    0    0]
37  take  -4  times  row  3  plus  row  1
38  take  -3/7  times  row  3  plus  row  2
```

```
39  [1  2  3  0]
40  [0  0  1  0]
41  [0  0  0  1]
42  [0  0  0  0]
43   take  -3  times  row  2  plus  row  1
44  [1  2  0  0]
45  [0  0  1  0]
46  [0  0  0  1]
47  [0  0  0  0]
```

These are naive implementations of Gauss's Method that do not try to work under stressful conditions, such as very small leading entries — they are just for fun. But they do illustrate perfectly the point made in the Preface, since a person builds intuition by doing a reasonable number of reasonably hard Gauss's Method reductions by hand, and at that point automation can take over.

Code comment: lines 50 and 88 are too long for this page so they end with a slash \ to make Python continue on the next line.

```
1  # code gauss_method.sage
2  # Show Gauss's method and Gauss-Jordan reduction steps.
3  # 2012-Apr-20   Jim Hefferon   Public Domain.
4
5  # Naive Gaussian reduction
6  def gauss_method(M, rescale_leading_entry=False):
7      """Describe the reduction to echelon form of the given matrix of
8      rationals.
9
10     M   matrix of rationals    e.g., M = matrix(QQ, [[..], [..], ..])
11     rescale_leading_entry=False  boolean  make leading entries to 1's
12
13     Returns: None.   Side effect: M is reduced.   Note that this is
14     echelon form, not reduced echelon form;, and that this routine
15     does not end the same way as does M.echelon_form().
16
17     """
18     num_rows=M.nrows()
19     num_cols=M.ncols()
20     print M
21
22     col = 0    # all cols before this are already done
23     for row in range(0,num_rows):
24         # ?Need to swap in a nonzero entry from below
25         while (col < num_cols
26                 and M[row][col] == 0):
27       for i in M.nonzero_positions_in_column(col):
28           if i > row:
29                   print " swap row",row+1,"with row",i+1
30         M.swap_rows(row,i)
31                   print M
32                   break
33             else:
34                 col += 1
35
```

```
36      if col >= num_cols:
37              break
38
39          # Now guaranteed M[row][col] != 0
40          if (rescale_leading_entry
41             and M[row][col] != 1):
42              print " take",1/M[row][col],"times row",row+1
43              M.rescale_row(row,1/M[row][col])
44        print M
45          change_flag=False
46          for changed_row in range(row+1,num_rows):
47              if M[changed_row][col] != 0:
48                  change_flag=True
49                  factor=-1*M[changed_row][col]/M[row][col]
50                  print " take",factor,"times row",row+1,   \
51                        "plus row",changed_row+1
52                  M.add_multiple_of_row(changed_row,row,factor)
53          if change_flag:
54              print M
55          col +=1
56
57 # Naive Gauss-Jordan reduction
58 def gauss_jordan(M):
59     """Describe the reduction to reduced echelon form of the
60     given matrix of rationals.
61
62     M   matrix of rationals   e.g., M = matrix(QQ, [[..], [..], ..])
63
64     Returns: None.   Side effect: M is reduced.
65
66     """
67     gauss_method(M,rescale_leading_entry=False)
68     # Get list of leading entries [le in row 0, le in row1, ..]
69     pivot_list=M.pivots()
70     # Rescale leading entries
71     change_flag=False
72     for row in range(0,len(pivot_list)):
73         col=pivot_list[row]
74         if M[row][col] != 1:
75             change_flag=True
76             print " take",1/M[row][col],"times row",row+1
77             M.rescale_row(row,1/M[row][col])
78     if change_flag:
79         print M
80     # Pivot
81     for row in range(len(pivot_list)-1,-1,-1):
82         col=pivot_list[row]
83         change_flag=False
84         for changed_row in range(0,row):
85             if M[changed_row,col] != 0:
86                 change_flag=True
```

```
87              factor=-1*M[changed_row][col]/M[row][col]
88              print " take",factor,"times row",   \
89                   row+1,"plus row",changed_row+1
90              M.add_multiple_of_row(changed_row,row,factor)
91      if change_flag:
92          print M
```

Vector Spaces

Sage can operate with vector spaces, for example by finding a basis for a space.

Real n-spaces

Start by creating a real vector space $\mathbb{R}^n$.

```
sage: V = RR^3
sage: V
Vector space of dimension 3 over Real Field with 53 bits of precision
```

You can test for membership.

```
sage: v1 = vector(RR, [1, 2, 3])
sage: v1 in V
True
sage: v2 = vector(RR, [1, 2, 3, 4])
sage: v2 in V
False
```

Consider the plane through the origin in $\mathbb{R}^3$ described by the equation $x - 2y + 2z = 0$. It is a subspace of $\mathbb{R}^3$ and we can describe it as a span.

$$W = \{ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \mid x = 2y - 2z \} = \{ \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} y + \begin{pmatrix} -2 \\ 0 \\ 1 \end{pmatrix} z \mid y, z \in \mathbb{R} \}$$

You can create that subspace.

```
sage: V = RR^3
sage: v1 = vector(RR, [2, 1, 0])
sage: v2 = vector(RR, [-2, 0, 1])
sage: W = V.span([v1, v2])
```

And you can check that it is the desired space by doing some membership tests.

```
sage: v3 = vector(RR, [0, 1, 1])
sage: v3 in W
True
sage: v4 = vector(RR, [1, 0, 0])
sage: v4 in W
False
```

For a closer look at what's happening here, create a subspace of $\mathbb{R}^4$. (Some of the output in this chapter is edited to fit the page width.)

```
1  sage: V = RR^4
2  sage: V
3  Vector space of dimension 4 over Real Field with 53 bits of precision
4  sage: v1 = vector(RR, [2, 0, -1, 0])
5  sage: W = V.span([v1])
6  sage: W
7  Vector space of degree 4 and dimension 1 over Real Field with 53 bits
8       of precision
9  Basis matrix:
10 [   1.000000000000000    0.000000000000000   -0.500000000000000
11       0.000000000000000]
```

Sage has identified the dimension of the subspace and found a basis containing one vector (it prefers the vector with a leading 1). Sage also mentions the "Real Field" because we could take scalars from other number systems — in the book's final chapter the scalars are from the complex numbers — but we will here stick with real numbers. The "53 bits of precision" refers to the fact that Sage is using this computer's built-in model of real numbers.[1]

As earlier, the membership set relation works here.

```
1  sage: v2 = vector(RR, [2, 1, -1, 0])
2  sage: v2 in W
3  False
4  sage: v3 = vector(RR, [-4, 0, 2, 0])
5  sage: v3 in W
6  True
```

Basis Sage will give you a basis for your space.

```
1  sage: V=RR^2
2  sage: v=vector(RR, [1, -1])
3  sage: W = V.span([v])
4  sage: W.basis()
5  [
6  (1.00000000000000, -1.00000000000000)
7  ]
```

Here is another example.

```
1  sage: V=RR^3
2  sage: v1 = vector(RR, [1, -1, 0])
3  sage: v2 = vector(RR, [1, 1, 0])
4  sage: W = V.span([v1, v2])
5  sage: W.basis()
6  [
7  (1.00000000000000, 0.000000000000000, 0.000000000000000),
8  (0.000000000000000, 1.00000000000000, 0.000000000000000)
9  ]
10 sage: W.basis()
```

---

[1]This computer uses IEEE 754 double-precision binary floating-point numbers; if you have programmed then you may know this number model as binary64.

```
11  [
12  (1.00000000000000,   0.000000000000000,   0.000000000000000),
13  (0.000000000000000,   1.00000000000000,   0.000000000000000)
14  ]
```

Adding a linearly dependent vector $\vec{v}_3$ to the spanning set doesn't change the space.

```
1  sage: v3 = vector(RR, [2, 3, 0])
2  sage: W_prime = V.span([v1, v2, v3])
3  sage: W_prime.basis()
4  [
5  (1.00000000000000,   0.000000000000000,   0.000000000000000),
6  (0.000000000000000,   1.00000000000000,   0.000000000000000)
7  ]
```

In the prior example, notice that Sage does not simply give you back the linearly independent vectors that you gave it. Instead, Sage takes the vectors from the spanning set as the rows of a matrix, brings that matrix to reduced echelon form, and reports the nonzero rows (transposed to column vectors and so printed with parentheses) as the members of the basis. Because each matrix has one and only one reduced echelon form, each vector subspace of real n-space has one and only one such basis; this is a *canonical basis* for the space.

Sage will show you that it keeps the canonical basis as defining the space.

```
1  sage: W
2  Vector  space  of  degree  3  and  dimension  2  over  Real  Field  with  53  bits
3       of  precision
4  Basis  matrix:
5  [ 1.00000000000000  0.000000000000000  0.000000000000000]
6  [0.000000000000000   1.00000000000000  0.000000000000000]
7  sage: W_prime
8  Vector  space  of  degree  3  and  dimension  2  over  Real  Field  with  53  bits
9       of  precision
10 Basis  matrix:
11 [ 1.00000000000000  0.000000000000000  0.000000000000000]
12 [0.000000000000000   1.00000000000000  0.000000000000000]
```

If you are quite keen on your own basis then Sage will accomodate you.

```
1  sage: V = RR^3
2  sage: v1 = vector(RR, [1, 2, 3])
3  sage: v2 = vector(RR, [2, 1, 3])
4  sage: W = V.span_of_basis([v1, v2])
5  sage: W.basis()
6  [
7  (1.00000000000000,   2.00000000000000,   3.00000000000000),
8  (2.00000000000000,   1.00000000000000,   3.00000000000000)
9  ]
10 sage: W
11 Vector  space  of  degree  3  and  dimension  2  over  Real  Field  with  53  bits
12      of  precision
13 User  basis  matrix:
14 [1.00000000000000  2.00000000000000  3.00000000000000]
15 [2.00000000000000  1.00000000000000  3.00000000000000]
```

Space equality   You can test spaces for equality.

```
1  sage:  V  =  RR^4
2  sage:  v1  =  vector(RR,  [1,  0,  0,  0])
3  sage:  v2  =  vector(RR,  [1,  1,  0,  0])
4  sage:  W12  =  V.span([v1,  v2])
5  sage:  v3  =  vector(RR,  [2,  1,  0,  0])
6  sage:  W13  =  V.span([v1,  v3])
```

Now use set membership to check that the two spaces W_{12} and W_{13} are equal.

```
1  sage:  v3  in  W12
2  True
3  sage:  v2  in  W13
4  True
```

Then, since obviously $\vec{v}_1 \in W_{12}$ and $\vec{v}_1 \in W_{13}$, the two spans are equal.
    You can also just ask Sage.

```
1  sage:  W12  ==  W13
2  True
```

Your check that equality works would be half-hearted if you didn't have an unequal case.

```
1   sage:  v4  =  vector(RR,  [1,  1,  1,  1])
2   sage:  W14  =  V.span([v1,  v4])
3   sage:  v2  in  W14
4   False
5   sage:  v3  in  W14
6   False
7   sage:  v4  in  W12
8   False
9   sage:  v4  in  W13
10  False
11  sage:  W12  ==  W14
12  False
13  sage:  W13  ==  W14
14  False
```

    This illustrates a point about algorithms. Sage could check for equality of two spans by checking whether every member of the first spanning set is in the second space and vice versa (since the two spanning sets are finite). But Sage does something different. For each space it maintains the canonical basis and it checks for equality of the two spaces just by checking that they have the same canonical bases. These two algorithms have the same external behavior, in that both report correctly whether the two spaces are equal, but the second is faster. Finding the fastest way to do jobs is an important research area of computing.

Space operations   Sage finds the intersection of two spaces. Consider these members of $\mathbb{R}^3$.

$$\vec{v}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{v}_3 = \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix}$$

We form two spans, the xy-plane $W_{12} = [\vec{v}_1, \vec{v}_2]$ and the yz-plane $W_{23} = [\vec{v}_2, \vec{v}_3]$. The intersection of these two is the y-axis.

```
1  sage:  V=RR^3
2  sage:  v1  =    vector(RR, [1,  0,  0])
3  sage:  v2  =    vector(RR, [0,  1,  0])
4  sage:  W12  =  V.span([v1,  v2])
5  sage:  W12.basis()
6  [
7  (1.00000000000000,  0.000000000000000,  0.000000000000000),
8  (0.000000000000000,  1.00000000000000,  0.000000000000000)
9  ]
10 sage:  v3  =  vector(RR, [0,  0,  2])
11 sage:  W23  =  V.span([v2,  v3])
12 sage:  W23.basis()
13 [
14 (0.000000000000000,  1.00000000000000,  0.000000000000000),
15 (0.000000000000000,  0.000000000000000,  1.00000000000000)
16 ]
17 sage:  W  =  W12.intersection(W23)
18 sage:  W.basis()
19 [
20 (0.000000000000000,  1.00000000000000,  0.000000000000000)
21 ]
```

Remember that the span of the empty set is the trivial space.

```
1  sage:  v3  =  vector(RR, [1,  1,  1])
2  sage:  W3  =  V.span([v3])
3  sage:  W3.basis()
4  [
5  (1.00000000000000,  1.00000000000000,  1.00000000000000)
6  ]
7  sage:  W4  =  W12.intersection(W3)
8  sage:  W4.basis()
9  [
10
11 ]
```

Sage will also find the sum of spaces, the span of their union.

```
1  sage:  W5  =  W12  +  W3
2  sage:  W5.basis()
3  [
4  (1.00000000000000,  0.000000000000000,  0.000000000000000),
5  (0.000000000000000,  1.00000000000000,  0.000000000000000),
6  (0.000000000000000,  0.000000000000000,  1.00000000000000)
7  ]
8  sage:  W5  ==  V
9  True
```

Other spaces

You can extend these computations to vector spaces that aren't a subspace of some $\mathbb{R}^n$. Just find a real space that matches the given one.

Consider this vector space set of quadratic polynomials (under the usual operations of polynomial addition and scalar multiplication).

$$\{a_2x^2 + a_1x + a_0 \mid a_2 - a_1 = a_0\} = \{(a_1 + a_0)x^2 + a_1x + a_0 \mid a_1, a_0 \in \mathbb{R}\}$$

It matches[1] this subspace of $\mathbb{R}^3$.

$$\{\begin{pmatrix} a_1 + a_0 \\ a_1 \\ a_0 \end{pmatrix} \mid a_1, a_0 \in \mathbb{R}\} = \{\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} a_1 + \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} a_0 \mid a_1, a_0 \in \mathbb{R}\}$$

```
1  sage:  V=RR^3
2  sage:  v1 = vector(RR, [1, 1, 0])
3  sage:  v2 = vector(RR, [1, 0, 1])
4  sage:  W = V.span([v1, v2])
5  sage:  W.basis()
6  [
7  (1.00000000000000, 0.000000000000000, 1.00000000000000),
8  (0.000000000000000, 1.00000000000000, -1.00000000000000)
9  ]
```

Similarly you can represent this space of $2{\times}2$ matrices

$$\{\begin{pmatrix} a & b \\ c & d \end{pmatrix} \mid a - b + c = 0 \text{ and } b + d = 0\}$$

by finding a real n-space just like it. Rewrite the given space

$$\{\begin{pmatrix} a & b \\ c & d \end{pmatrix} \mid a = -c - d \text{ and } b = -d\} = \{\begin{pmatrix} -1 & 0 \\ 1 & 0 \end{pmatrix} c + \begin{pmatrix} -1 & -1 \\ 0 & 1 \end{pmatrix} d \mid c, d \in \mathbb{R}\}$$

and then here is a natural matching real space.

```
1  sage:  V = RR^4
2  sage:  v1 = vector(RR, [-1, 0, 1, 0])
3  sage:  v2 = vector(RR, [-1, -1, 0, 1])
4  sage:  W = V.span([v1, v2])
5  sage:  W.basis()
6  [
7  (1.00000000000000, -0.000000000000000, -1.00000000000000,
8      -0.000000000000000),
9  (0.000000000000000, 1.00000000000000, 1.00000000000000,
10     -1.00000000000000)
11 ]
```

You could have gotten another matching space by going down the columns instead of across the rows, etc. But the important things about the space, such as its dimension, are unaffected by that choice.

---

[1] The textbook's chapter on Maps Between Spaces makes "matches" precise.

Matrices

Matrix operations are mechanical, and are therefore perfectly suited for mechanizing.

Defining

To define a matrix you can use real number entries, or complex entries, or entries from other number systems such as the rationals.

```
sage: A = matrix(RR, [[1, 2], [3, 4]])
sage: A
[1.00000000000000  2.00000000000000]
[3.00000000000000  4.00000000000000]
sage: i = CC(i)
sage: A = matrix(CC, [[1+2*i, 3+4*i], [5+6*i, 7+8*i]])
sage: A
[1.00000000000000 + 2.00000000000000*I
     3.00000000000000 + 4.00000000000000*I]
[5.00000000000000 + 6.00000000000000*I
     7.00000000000000 + 8.00000000000000*I]
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: A
[1 2]
[3 4]
```

(Sometimes, as here, the output is edited to fit the page. Note that before working with complex numbers we resetting i to be $\sqrt{-1}$. The letter i is used for many things so resetting is a good habit.) Unless we have a reason to do otherwise, we'll use rational numbers because the matrices easier to read.

The `matrix` constructor allows you to specify the number of rows and columns.

```
sage: B = matrix(QQ, 2, 3, [[1, 1, 1], [2, 2, 2]])
sage: B
[1 1 1]
[2 2 2]
```

If your specified size doesn't match the entries

```
sage: B = matrix(QQ, 3, 3, [[1, 1, 1], [2, 2, 2]])
```

then Sage's error says `Number of rows does not match up with specified number`. Until now we've let Sage figure out the matrix's number of rows and columns size from the entries but a shortcut to get the zero matrix is to put the number zero in the place of the entries, and there you must say which size you want.

```
1  sage: B = matrix(QQ, 2, 3, 0)
2  sage: B
3  [0 0 0]
4  [0 0 0]
```

Another case is the shortcut to get an identity matrix.

```
1  sage: B = matrix(QQ, 2, 2, 1)
2  sage: B
3  [1 0]
4  [0 1]
```

The difference with this case is that matrix(QQ, 3, 2, 1) gives an error because an identity matrix must be square. Sage has another shortcut that can't lead to this error.

```
1  sage: I = identity_matrix(4)
2  sage: I
3  [1 0 0 0]
4  [0 1 0 0]
5  [0 0 1 0]
6  [0 0 0 1]
```

Sage has a wealth of methods on matrices. For instance, you can transpose the rows to columns or test if the matrix is *symmetric*, unchanged by transposition.

```
1  sage: A.transpose()
2  [1 3]
3  [2 4]
4  sage: A.is_symmetric()
5  False
```

Linear combinations

Addition and subtraction are natural.

```
1  sage: B = matrix(QQ, [[1, 1], [2, -2]])
2  sage: A+B
3  [2 3]
4  [5 2]
5  sage: A-B
6  [0 1]
7  [1 6]
8  sage: B-A
9  [ 0 -1]
10 [-1 -6]
```

Sage won't let you combine matrices with different sizes; this gives an error.

```
1  sage: C = matrix(QQ, [[0, 0, 2], [3, 2, 1]])
2  sage: A+C
```

The last line of the error says TypeError: unsupported operand parent(s) for '+' and describes the problem, albiet in somewhat technical terms, as that you can't add a $2{\times}2$ matrix to a $2{\times}3$ matrix.

Scalar multiplication is also natural so you have linear combinations.

```
1  sage: 3*A
2  [ 3   6]
3  [ 9  12]
4  sage: 3*A-4*B
5  [-1   2]
6  [ 1  20]
```

Multiplication

Matrix-vector product   Matrix-vector multiplication is just what you would guess.

```
1  sage: A = matrix(QQ, [[1, 3, 5, 9], [0, 2, 4, 6]])
2  sage: v = vector(QQ, [1, 2, 3, 4])
3  sage: A*v
4  (58, 40)
```

Here the 2×4 matrix A multiplies the 4×1 column vector $\vec{v}$, with the vector on the right side, as $A\vec{v}$. If you try this vector on the left as v*A then Sage gives an error saying the sizes don't match.

You can make a vector to give a defined multiplcation from the left side.

```
1  sage: w = vector(QQ, [3, 5])
2  sage: w*A
3  (3, 19, 35, 57)
```

In practice you see the vector on either side. We will have the habit of putting the vector on the right because that is what's in the book.

Matrix-matrix product   If the sizes match then Sage will multiply the matrices in the obvious way. Here is the product of a 2×2 matrix A and a 2×3 matrix B.

```
1  sage: A = matrix(QQ, [[2, 1], [4, 3]])
2  sage: B = matrix(QQ, [[5, 6, 7], [8, 9, 10]])
3  sage: A*B
4  [18 21 24]
5  [44 51 58]
```

Trying B*A gives the error unsupported operand parent(s) for '*', meaning that the product operation in this order is undefined.

Same-sized square matrices have the product defined in either order.

```
1  sage: A = matrix(QQ, [[1, 2], [3, 4]])
2  sage: B = matrix(QQ, [[4, 5], [6, 7]])
3  sage: A*B
4  [16 19]
5  [36 43]
6  sage: B*A
7  [19 28]
8  [27 40]
```

Of course they are different; matrix multiplication is not commutative.

```
1  sage: A*B == B*A
2  False
```

It is very noncommutative, in that if you produce two $n \times n$ matrices at random then they almost surely don't commute.

```
1  sage: random_matrix(RR, 3, min=-1, max=1)
2  [ 0.316363590763196    0.401338054275907   -0.434566166245099]
3  [ 0.971906850779329   -0.189878350367539    0.0883946411446965]
4  [-0.892505949015383   -0.976541221718267   -0.278783066662399]
5  sage: random_matrix(RR, 3, min=-1, max=1)
6  [ 0.993363158808678    0.839878618006500   -0.431488448252715]
7  [-0.161186636590872    0.554717067849706   -0.414238110166975]
8  [-0.524416310634741   -0.897687685090845    0.829719717481328]
```

(Note the RR. We prefer real number entries here because random_matrix is more straightforward in this case than in the rational entry case.)

```
1  sage: number_commuting = 0
2  sage: for n in range(1000):
3  ....:       A = random_matrix(RR, 2, min=-1, max=1)
4  ....:       B = random_matrix(RR, 2, min=-1, max=1)
5  ....:       if (A*B == B*A):
6  ....:               number_commuting = number_commuting + 1
7  ....:
8  sage: print "number commuting of 1000=", number_commuting
9  number commuting of 1000= 0
```

Inverse Recall that if A is nonsingular then its *inverse* A^{-1} is the matrix such that $AA^{-1} = A^{-1}A$ is the identity matrix.

```
1  sage: A = matrix(QQ, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
2  sage: A.is_singular()
3  False
```

We have a formula for 2×2 matrix inverses but to compute inverses for larger matrices we write the original matrix next to the identity, and then do Gauss-Jordan reduction.

```
1  sage: I = identity_matrix(3)
2  sage: B = A.augment(I, subdivide=True)
3  sage: B
4  [ 1   3   1| 1   0   0]
5  [ 2   1   0| 0   1   0]
6  [ 4  -1   0| 0   0   1]
7  sage: C = B.rref()
8  sage: C
9  [      1       0       0|       0     1/6     1/6]
10 [      0       1       0|       0     2/3    -1/3]
11 [      0       0       1|       1   -13/6     5/6]
```

The inverse is the resulting matrix on the right.

```
1  sage: A_inv = C.matrix_from_columns([3, 4, 5])
2  sage: A_inv
3  [      0     1/6     1/6]
4  [      0     2/3    -1/3]
5  [      1   -13/6     5/6]
```

```
6  sage: A*A_inv
7  [1  0  0]
8  [0  1  0]
9  [0  0  1]
10 sage: A_inv*A
11 [1  0  0]
12 [0  1  0]
13 [0  0  1]
```

Since this is an operation that Sage users do all the time, there is a standalone command.

```
1  sage: A_inv = A.inverse()
2  sage: A_inv
3  [    0    1/6    1/6]
4  [    0    2/3   -1/3]
5  [    1 -13/6    5/6]
```

One reason for finding the inverse is to make solving linear systems easier. Consider these.

$$
\begin{array}{lll}
\begin{aligned}
x + 3y + z &= 4 \\
2x + y &= 4 \\
4x - y &= 4
\end{aligned}
&
\begin{aligned}
x + 3y + z &= 2 \\
2x + y &= -1 \\
4x - y &= 5
\end{aligned}
&
\begin{aligned}
x + 3y + z &= 1/2 \\
2x + y &= 0 \\
4x - y &= 12
\end{aligned}
\end{array}
$$

They share the matrix of coefficents A but have different vectors on the right side. Having calculated the inverse of the matrix of coefficients, solving each system takes just a matrix-vector product.

```
1  sage: v1 = vector(QQ, [4, 4, 4])
2  sage: v2 = vector(QQ, [2, -1, 5])
3  sage: v3 = vector(QQ, [1/2, 0, 12])
4  sage: A_inv*v1
5  (4/3,  4/3,  -4/3)
6  sage: A_inv*v2
7  (2/3,  -7/3,  25/3)
8  sage: A_inv*v3
9  (2,  -4,  21/2)
```

Run time

Since computers are fast and accurate they open up the possibility of solving problems that are quite large. Large linear algebra problems occur frequently in science and engineering. In this section we will suggest what limits there are to computers. (Here we will use matrices with real entries because they are common in applications.)

One of the limits on just how large a problem we can do is how quickly the computer program can give answers. Naturally computers take longer to perform operations on matrices that are larger but it may be that the time the program takes to compute the answer grows more quickly than does the size of the problem — for instance, when the size of the matrix doubles then the time to do the job more than doubles.

We will time the matrix inverse operation. This is an important operation; for instance, if we could do large matrix inverses quickly then we could quickly solve large linear systems, with just a matrix-vector product. We also use it because it makes a good illustration.

```
1  sage: A = matrix(RR, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
2  sage: A
3  [  1.00000000000000   3.00000000000000   1.00000000000000]
4  [  2.00000000000000   1.00000000000000   0.000000000000000]
5  [  4.00000000000000  -1.00000000000000   0.000000000000000]
6  sage: A.is_singular()
7  False
8  sage: timeit('A.inverse()')
9  625 loops, best of 3: 124 µs per loop
```

It took 124 microseconds, which is $0.000\,124$ seconds.[1] That's fast, but then A is only a 3×3 matrix.

To scale up the timing, you can find the inverse of a random matrix.

```
1  sage: timeit('random_matrix(RR, 3, min=-1, max=1).inverse()')
2  625 loops, best of 3: 186 µs per loop
```

This has the issue that we can't tell right away whether the time is spent generating the random matrix or finding the inverse. We also can't tell whether this command generates many random matrices and finds each's inverse, or generates one matrix and find its inverse many times. This code is clearer.

```
1  sage: for size in [3, 10, 25, 50, 75, 100, 150, 200]:
2  ....:     print "size=", size
3  ....:     M = random_matrix(RR, size, min=-1, max=1)
4  ....:     timeit('M.inverse()')
5  ....:
6  size= 3
7  625 loops, best of 3: 125 µs per loop
8  size= 10
9  625 loops, best of 3: 940 µs per loop
10 size= 25
11 25 loops, best of 3: 12 ms per loop
12 size= 50
13 5 loops, best of 3: 92.4 ms per loop
14 size= 75
15 5 loops, best of 3: 308 ms per loop
16 size= 100
17 5 loops, best of 3: 727 ms per loop
18 size= 150
19 5 loops, best of 3: 2.45 s per loop
20 size= 200
21 5 loops, best of 3: 5.78 s per loop
```

Some of those times are in microseconds, some are in milliseconds, and some are in seconds. This table is consistently in seconds.

---

[1] Sage runs the command many times and comes up with a best guess about how long the operation ideally takes, because on any one time your computer may have been slowed down by a disk write or some other interruption.

size	seconds
3	0.000 125
10	0.000 940
25	0.012
50	0.0924
75	0.308
100	0.727
150	2.45
200	5.78

The time grows faster than the size. For instance, in going from size 25 to size 50 the time more than doubles: $0.0924/0.012$ is 7.7. Similarly, increasing the size from 50 to 200 causes the time to increase by much more than a factor of four: $5.78/0.0924 \approx 62.55$.
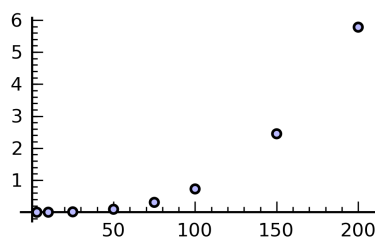
To get a picture give Sage the data as a list of pairs.

```
sage: d = [(3, 0.000125), (10, 0.000940), (25, 0.012),
....:       (50, 0.0924), (75, 0.308), (100, 0.727),
....:       (150, 2.45), (200, 5.78)]
sage: g = scatter_plot(d)
sage: g.save("inverse_initial.png")
```

(If you enter `scatter_plot(d)` without saving it as `g` then Sage will pop up a window with the graphic.)[1]

The graph dramatizes that the the ratio time/size is not constant since the data clearly does not look like a line.

Here is some more data. The times are big enough that the computer had to run overnight.

```
sage: for size in [500, 750, 1000]:
....:       print "size=", size
....:       M = random_matrix(RR, size, min=-1, max=1)
....:       timeit('M.inverse()')
....:
size= 500
5 loops, best of 3: 89.2 s per loop
size= 750
5 loops, best of 3: 299 s per loop
size= 1000
5 loops, best of 3: 705 s per loop
```

---

[1]The graphics here use a few of Sage's options to make them better fit this page. For instance, the first graphic was generated with `g1 = scatter_plot(d, markersize=10, facecolor='#b9b9ff')`. It was saved with `g1.save("../inverse_initial.png", figsize=[2.25,1.5], axes_pad=0.05, fontsize=7, dpi=500)`.
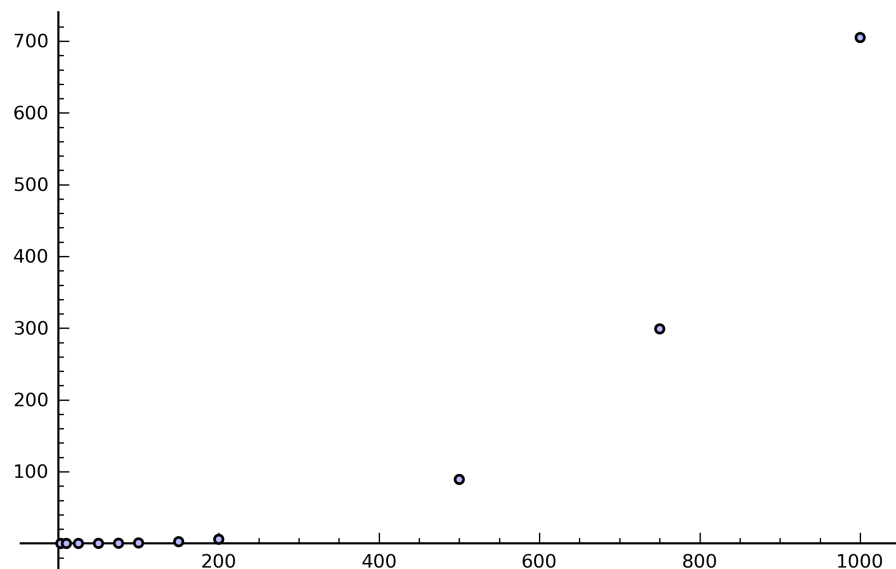
Again the table is neater.

size	seconds
500	89.2
750	299.
1000	705.

Get a graph by tacking the new data onto the existing data.

```
sage: d = d + [(500, 89.2), (750, 299), (1000, 705)]
sage: g = scatter_plot(d)
sage: g.save("inverse_full.png")
```

The result is this graphic.

Note that the two graphs have different scales. For one thing, if the vertical height for 6 was as large on the second graph as on the first then the data would go far off the top of the page.

So a practical limit to the size of a problem that we can solve with the matrix inverse operation comes from the fact that the graph above is not a line. Beyond some size the time required just gets too large.

Maps

We've used Sage to define vector spaces. Next we explore operations that we can do on vector spaces.

Left/right

Sage represents linear maps differently than the book does. Rather than quarrel with the tool, below we will do it Sage's way.

Consider the application of a linear map to a member of a vector space $t(\vec{v})$. For example, with this function $t\colon \mathbb{R}^2 \to \mathbb{R}^3$ and this element of the domain

$$t(\begin{pmatrix} a \\ b \end{pmatrix}) = \begin{pmatrix} a+b \\ a-b \\ b \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

the map application gives this.

$$t(\begin{pmatrix} 1 \\ 3 \end{pmatrix}) = \begin{pmatrix} 4 \\ -2 \\ 3 \end{pmatrix}$$

To represent the map application we first fix bases. In this example we use the canonical bases $E_2 \subset \mathbb{R}^2$ and $E_3 \subset \mathbb{R}^3$. Then with respect to the bases the book finds a matrix $T = \mathrm{Rep}_{E_2, E_3}(t)$ and a column vector $\vec{w} = \mathrm{Rep}_{E_2}(\vec{v})$, and represents $t(\vec{v})$ with the product $T\vec{w}$.

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \end{pmatrix} = \begin{pmatrix} 4 \\ -2 \\ 3 \end{pmatrix}$$

That is, the book is write right: its notation puts the vector on the right of the matrix.

However, this choice is a matter of taste and many authors instead use a row vector that multiplies a matrix from the left. Sage is in this camp and represents the map application in this way.

$$(1 \quad 3) \begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} = (4 \quad -2 \quad 3)$$

Obviously the difference is cosmetic but can cause confusion. The translation is that, compared to the book's representation $T\vec{w}$, Sage prefers the transpose $(T\vec{w})^\mathsf{T} = \vec{w}^\mathsf{T}\, T^\mathsf{T}$.

Defining

We will see two different ways to define a linear transformation.

Symbolically   We first define a map that takes two inputs and returns three outputs.

```
1  sage: a, b = var('a, b')
2  sage: T_symbolic(a, b) = [a+b, a-b, b]
3  sage: T_symbolic
4  (a, b) |--> (a + b, a - b, b)
```

We have not yet defined a domain and codomain so this not a function—instead it is a prototype for a function. Make an instance of a function by applying $T_{symbolic}$ on a particular domain and codomain.

```
1  sage: T = linear_transformation(RR^2, RR^3, T_symbolic)
2  sage: T
3  Vector space morphism represented by the matrix:
4  [  1.00000000000000   1.00000000000000  0.000000000000000]
5  [  1.00000000000000  -1.00000000000000   1.00000000000000]
6  Domain: Vector space of dimension 2 over Real Field with 53 bits of
7                                                            precision
8  Codomain: Vector space of dimension 3 over Real Field with 53 bits of
9                                                            precision
```

Note the left/right issue again: Sage's matrix is the transpose of the matrix that the book would use.

Evaluating this function on a member of the domain gives a member of the codomain.

```
1  sage: v = vector(RR, [1, 3])
2  sage: T(v)
3  (4.00000000000000, -2.00000000000000, 3.00000000000000)
```

Sage can compute the interesting things about the transformation. Here it finds the null space and range space, using the equivalent terms *kernel* and *image*.

```
1  sage: T.kernel()
2  Vector space of degree 2 and dimension 0 over Real Field with 53 bits
3                                                         of precision
4  Basis matrix:
5  []
6  sage: T.image()
7  Vector space of degree 3 and dimension 2 over Real Field with 53 bits
8                                                         of precision
9  Basis matrix:
10 [   1.00000000000000   0.000000000000000   0.500000000000000]
11 [  0.000000000000000    1.00000000000000  -0.500000000000000]
```

The null space's basis is empty because it is the trivial subspace of the domain, with dimension 0. Therefore T is one-to-one.

The range space has a 2-vector basis. This agrees with the theorem that the dimension of the null space plus the dimension of the range space equals to the dimension of the domain.

For contrast consider a map that is not one-to-one.

```
1  sage:  S_symbolic(a,  b)  =  [a+2*b,  a+2*b]
2  sage:  S_symbolic
3  (a,  b)  |-->  (a  +  2*b,  a  +  2*b)
4  sage:  S  =  linear_transformation(RR^2,  RR^2,  S_symbolic)
5  sage:  S
6  Vector  space  morphism  represented  by  the  matrix:
7  [1.00000000000000  1.00000000000000]
8  [2.00000000000000  2.00000000000000]
9  Domain:  Vector  space  of  dimension  2  over  Real  Field  with  53  bits  of
10                                                                   precision
11  Codomain:  Vector  space  of  dimension  2  over  Real  Field  with  53  bits  of
12                                                                   precision
13  sage:  v  =  vector(RR,  [1,  3])
14  sage:  S(v)
15  (7.00000000000000,  7.00000000000000)
```

This map is not one-to-one since the input $(a, b) = (2, 0)$ gives the same result as $(a, b) = (0, 1)$.

```
1  sage:  S.kernel()
2  Vector  space  of  degree  2  and  dimension  1  over  Real  Field  with  53  bits
3                                                                   of  precision
4  Basis  matrix:
5  [   1.00000000000000   -0.500000000000000]
6  sage:  S.image()
7  Vector  space  of  degree  2  and  dimension  1  over  Real  Field  with  53  bits
8                                                                   of  precision
9  Basis  matrix:
10  [1.00000000000000  1.00000000000000]
```

The null space has nonzero dimension, namely it has dimension 1, so Sage agrees that the map is not one-to-one.

Without looking at the range space we know that its dimension must be 1 because the dimensions of the null and range spaces add to the dimension of the domain. Again, Sage confirms our calculation.

Via matrices We can define a transformation by specifying the matrix representing its action.

```
1  sage:  M  =  matrix(RR,  [[1,  2],  [3,  4],  [5,  6]])
2  sage:  m  =  linear_transformation(M)
3  sage:  m
4  Vector  space  morphism  represented  by  the  matrix:
5  [1.00000000000000  2.00000000000000]
6  [3.00000000000000  4.00000000000000]
7  [5.00000000000000  6.00000000000000]
8  Domain:  Vector  space  of  dimension  3  over  Real  Field  with  53  bits  of
9                                                                   precision
10  Codomain:  Vector  space  of  dimension  2  over  Real  Field  with  53  bits  of
11                                                                   precision
```

Note again that Sage prefers the representation where the vector multiplies from the left.

```
1  sage:  v  =  vector(RR,  [7,  8,  9])
2  sage:  m(v)
3  (76.0000000000000,  100.000000000000)
```

Sage has done this calculation.

$$\begin{pmatrix} 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 76 & 100 \end{pmatrix}$$

If you have a matrix intended for a vector-on-the-right calculation (as in the book) then Sage will make the necessary adaptation.

```
sage: N = matrix(RR, [[1, 3, 5], [2, 4, 6]])
sage: n = linear_transformation(N, side='right')
sage: n
Vector space morphism represented by the matrix:
[1.00000000000000  2.00000000000000]
[3.00000000000000  4.00000000000000]
[5.00000000000000  6.00000000000000]
Domain: Vector space of dimension 3 over Real Field with 53 bits of
                                                          precision
Codomain: Vector space of dimension 2 over Real Field with 53 bits of
                                                          precision
sage: v = vector(RR, [7, 8, 9])
sage: n(v)
(76.0000000000000, 100.000000000000)
```

Although we gave it a `side='right'` option, the matrix that Sage shows by default is for `side='left'`.

Despite that we specified them differently, these two transformations are the same.

```
sage: m == n
True
```

We can ask the same questions of linear transformations created from matrices that we asked of linear transformations created from functions.

```
sage: m.kernel()
Vector space of degree 3 and dimension 1 over Real Field with 53 bits
                                                     of precision
Basis matrix:
[ 1.00000000000000  -2.00000000000000  1.00000000000000]
```

The null space of m is not the trivial subspace of $\mathbb{R}^3$ and so this function is not one-to-one. The domain has dimension 3 and the null space has dimension 1 and so the range space is a dimension 2 subspace of $\mathbb{R}^2$.

```
sage: m.image()
Vector space of degree 2 and dimension 2 over Real Field with 53 bits
                                                     of precision
Basis matrix:
[ 1.00000000000000  0.000000000000000]
[0.000000000000000  1.00000000000000]
sage: m.image() == RR^2
True
```

Sage lets us have the matrix represent a transformation involving spaces with nonstandard bases.

```
1  sage: M = matrix(RR, [[1, 2], [3, 4]])
2  sage: domain_basis = [vector(RR, [1, -1]), vector(RR, [1, 1])]
3  sage: D = (RR^2).subspace_with_basis(domain_basis)
4  sage: codomain_basis = [vector(RR, [2, 0]), vector(RR, [0, 3])]
5  sage: C = (RR^2).subspace_with_basis(codomain_basis)
6  sage: m = linear_transformation(D, C, M)
7  sage: m(vector(RR, [1, 0]))
8  (4.00000000000000, 9.00000000000000)
```

Sage has calculated that

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = (1/2)\begin{pmatrix} 1 \\ -1 \end{pmatrix} + (1/2)\begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \text{so} \quad \mathrm{Rep}_{domain\_basis}(\begin{pmatrix} 1 \\ 0 \end{pmatrix}) = \begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix}$$

and then computed this.

$$\mathrm{Rep}_{codomain\_basis}(m(\vec{v})) = (1/2 \quad 1/2)\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = (2 \quad 3) \quad \text{so} \quad m(\vec{v}) = 2\begin{pmatrix} 2 \\ 0 \end{pmatrix} + 3\begin{pmatrix} 0 \\ 3 \end{pmatrix} = \begin{pmatrix} 4 \\ 9 \end{pmatrix}$$

Operations

Consider the set of linear maps from some Fix some vector space domain D and codomain C and consider the set of all linear transformations between them. This set has some natural operations, including addition and scalar multiplication. Sage can work with those operations.

Addition   Recall that matrix addition is defined so that the representation of the sum of two linear transformations is the matrix sum of the representatives. Sage can illustrate.

```
1  sage: M = matrix(RR, [[1, 2], [3, 4]])
2  sage: m = linear_transformation(RR^2, RR^2, M)
3  sage: m
4  Vector space morphism represented by the matrix:
5  [1.00000000000000 2.00000000000000]
6  [3.00000000000000 4.00000000000000]
7  Domain: Vector space of dimension 2 over Real Field with 53 bits of
8                                                            precision
9  Codomain: Vector space of dimension 2 over Real Field with 53 bits of
10                                                           precision
11 sage: N = matrix(RR, [[5, -1], [0, 7]])
12 sage: n = linear_transformation(RR^2, RR^2, N)
13 sage: n
14 Vector space morphism represented by the matrix:
15 [ 5.00000000000000  -1.00000000000000]
16 [0.000000000000000   7.00000000000000]
17 Domain: Vector space of dimension 2 over Real Field with 53 bits of
18                                                           precision
19 Codomain: Vector space of dimension 2 over Real Field with 53 bits of
20                                                           precision
21 sage: m+n
22 Vector space morphism represented by the matrix:
23 [6.00000000000000 1.00000000000000]
```

```
24 [3.00000000000000  11.0000000000000]
25 Domain: Vector space of dimension 2 over Real Field with 53 bits of
26                                                             precision
27 Codomain: Vector space of dimension 2 over Real Field with 53 bits of
28                                                             precision
29 sage: M+N
30 [6.00000000000000  1.00000000000000]
31 [3.00000000000000  11.0000000000000]
```

Similarly, linear map scalar multiplication is reflected by matrix scalar multiplication.

```
1 sage: m*3
2 Vector space morphism represented by the matrix:
3 [3.00000000000000  6.00000000000000]
4 [9.00000000000000  12.0000000000000]
5 Domain: Vector space of dimension 2 over Real Field with 53 bits of
6                                                             precision
7 Codomain: Vector space of dimension 2 over Real Field with 53 bits of
8                                                             precision
9 sage: M*3
10 [3.00000000000000  6.00000000000000]
11 [9.00000000000000  12.0000000000000]
```

Composition   The composition of linear maps gives rise to matrix multiplication. Sage uses the star ∗ to denote composition of linear maps.

```
1 sage: M = matrix(RR, [[1, 2], [3, 4]])
2 sage: m = linear_transformation(RR^2, RR^2, M)
3 sage: m
4 Vector space morphism represented by the matrix:
5 [1.00000000000000  2.00000000000000]
6 [3.00000000000000  4.00000000000000]
7 Domain: Vector space of dimension 2 over Real Field with 53 bits of
8                                                             precision
9 Codomain: Vector space of dimension 2 over Real Field with 53 bits of
10                                                             precision
11 sage: N = matrix(RR, [[5, -1], [0, 7]])
12 sage: n = linear_transformation(RR^2, RR^2, N)
13 sage: n
14 Vector space morphism represented by the matrix:
15 [ 5.00000000000000  -1.00000000000000]
16 [0.000000000000000   7.00000000000000]
17 Domain: Vector space of dimension 2 over Real Field with 53 bits of
18                                                             precision
19 Codomain: Vector space of dimension 2 over Real Field with 53 bits of
20                                                             precision
21 sage: M*N
22 [5.00000000000000  13.0000000000000]
23 [15.0000000000000  25.0000000000000]
24 sage: N*M
25 [2.00000000000000  6.00000000000000]
26 [21.0000000000000  28.0000000000000]
```

```
27  sage: m*n
28  Vector space morphism represented by the matrix:
29  [2.00000000000000  6.00000000000000]
30  [21.0000000000000  28.0000000000000]
31  Domain: Vector space of dimension 2 over Real Field with 53 bits of
32                                                              precision
33  Codomain: Vector space of dimension 2 over Real Field with 53 bits of
34                                                              precision
```

Note the left/right issue. Remember that $m \circ n$ is the map $\vec{v} \mapsto m(n(\vec{v}))$, so that n is applied first. Sage puts the representing vector on the left, so N must come left-most: $\vec{w}\, N \cdot M = \vec{w}(NM)$.

Bibliography

Robert A. Beezer. Sage for Linear Algebra. http://linear.ups.edu/download/fcla-2.22-sage-4.7.1-preview.pdf, 2011.

Jim Hefferon. Linear Algebra. http://joshua.smcvt.edu/linearalgebra, 2012.

David Joyner and William Stein. Open source mathematical software. *Notices of the AMS*, page 1279, November 2007.

Sage Development Team. Sage tutorial 5.3. http://www.sagemath.org/pdf/SageTutorial.pdf, 2012a.

Sage Development Team. Sage reference manual 5.3. http://www.sagemath.org/pdf/reference.pdf, 2012b.